A precision and range independent tool for testing floating-point arithmetic I : basic operations, square root and remainder

Brigitte Verdonk,
Annie Cuyt
and
Dennis Verschaeren

University of Antwerp, Department of Mathematics and Computer Science, Universiteitsplein 1, B-2610 Antwerp, Belgium.
e-mail: verdonk@uia.ua.ac.be

This paper introduces a precision and range independent tool for testing the compliance of hardware or software implementations of (multiprecision) floating-point arithmetic with the principles of the IEEE standards 754 and 854. The tool consists of a driver program, offering many options to test only specific aspects of the IEEE standards, and a large set of test vectors, encoded in a precision independent syntax to allow the testing of basic and extended hardware formats as well as multiprecision floating-point implementations.

The suite of test vectors stems on one hand from the integration and fully precision and range independent generalisation of existing hardware test sets, and on the other hand from the systematic testing of exact rounding for all combinations of round and sticky bit that can occur. The former constitutes only 50% of the resulting test set. In the latter we especially focus on hard to round cases.

In addition, the test suite implicitly tests properties of floating-point operations, following the idea of Paranoia, and it reports which of the three IEEE-compliant underflow mechanisms is used by the floating-point implementation under consideration. We also check whether that underflow mechanism is used consistently.

The tool is backward compatible with the UCBTEST package and with Coonen's test syntax.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; D.3.0 [**Programming Languages**]: Processors—*Compilers*; D.3.0 [**Programming Languages**]: General—*Standards*

General Terms: floating-point, arithmetic

Additional Key Words and Phrases: multiprecision, IEEE floating-point standard, verification, validation

1. INTRODUCTION AND MOTIVATION

The IEEE standard [IEEE 1985] for floating-point arithmetic, which became official in 1985 and which we shall refer to as IEEE–754, has been adopted by most major microprocessor manufacturers. Whereas guaranteeing 100% correctness of an IEEE floating-point implementation is hardly feasible, as the famous Intel Pentium bug clearly demonstrated, several good but unrelated tools exist to test different aspects of a floating-point implementation for compliance with the IEEE–754 standard.

Concurrent with the adoption of IEEE–754 by the microprocessor industry, there has been a growing need in many applications for more precision than provided by the IEEE single and double formats. According to W. Kahan, for now the 10-byte extended hardware format is a tolerable compromise between the value of extra-precise arithmetic and the price of implementing it to run fast; very soon 2 more bytes of precision will become tolerable, and ultimately a 16-byte format ...

That kind of gradual evolution towards wider precision, was already in view when the IEEE standards 754-854 were framed [Higham 1996, p. 47]. Furthermore, predictions based on the growth in the size of mathematical models solved as the memory and speed of computers increase, suggest that floating-point arithmetic with unit roundoff $\approx 10^{-32}$ will be needed for some applications on future supercomputers [Bailey et al. 1989].

Against this background, 16-byte format floating-point arithmetic [SUN Microsystems 1997] as well as several multiprecision implementations have been developed in the last decades. Unfortunately, tools for testing such implementations are much more limited than tools for testing single and double precision floating-point arithmetic. For example, for existing multiprecision floating-point packages such as [Brent 1978], [Moshier 1989], [Smith 1991], [Bailey 1995], [Batut et al.], [Granlund 2000], [Tommila 2000], [Haible] and [Zimmermann et al. 2000], standard testing techniques consist of internal consistency checks such as $\sqrt{x^2} = |x|$, or comparison of low, respectively high precision results with values obtained using hardware formats, or other multiprecision packages, and this for randomly chosen operands. This randomly chosen subset does not guarantee that all essential aspects of a particular operation are tested.

In this report, we describe a comprehensive, precision and range independent tool to test how well a floating-point implementation complies with the philosophy of the IEEE–754 and IEEE–854 [IEEE 1987] standards. These standards list, besides requirements concerning floating-point formats and specifications for rounding, also specifications for

(a) add, subtract, multiply, divide, square root, remainder and compare operations
(b) conversions between different floating-point formats
(c) conversions between integer and floating-point formats
(d) rounding of floating-point numbers to integral value
(e) conversions between basic format floating-point numbers and decimal strings
(f) floating-point exceptions and their handling, including nonnumbers (NaNs)

that can easily be formulated in a format independent way. Our tool can be used to test how well these principles are met in an arithmetic implementation for floating-point formats with arbitrary precision and exponent range. The tested

floating-point system can be implemented in hardware, in software or a combination of both. The only requirement is that the base $\beta = 2$. For testing multiprecision software packages, this is not really a restriction since in most of these packages the base β can be specified by the user, within certain bounds. Clearly, even though our test tool contains many tough cases and is certainly better than a random testing strategy, an implementation that passes all aspects of the test without error is not guaranteed to be 100% correct. On the other hand, our test tool has been used to check a number of floating-point implementations and was able to detect anomalies that had passed unnoticed before.

The precision independent tool we have developed is designed to test a floating-point system in its globality, as a programming environment, in other words all operations (a), all conversions (b) through (e), as well as the handling of all floating-point exceptions (f). In this paper, we only present those aspects of the test tool that check the operations add, subtract, multiply, divide, square root and remainder, including the floating-point exceptions raised by those operations. In a forthcoming paper [Verdonk et al. 2000], we shall describe the testing of all conversions listed in (b) through (e). The splitting up of the description in two distinct parts is motivated by several factors:

- First, for most microprocessors it is the case that at least part of the floating-point environment and certainly the operations (a), are implemented in hardware. On the other hand, some of the conversions, such as decimal-binary conversions, are provided only by the compiler.
- An even more substantial difference between the operations $+, -, \times, /, \sqrt{\ }$ and the other aspects of a hardware floating-point environment, concerns the binding between the operators at the language level and the underlying implementation. For $+, -, \times, /, \sqrt{\ }$, this binding is unambiguous, while for remainder and conversions it may be compiler and/or programming language specific. Furthermore, those operations which are available in hardware and are IEEE compliant, may be inaccessible to the programmer due to programming language specifications. Hence the distinction between what is actually being tested, the language binding or the effectively available implementation, becomes more and more essential.
- It is also the case that, whereas the standard is very strict for the operations (a) and requires exact rounding for all arithmetic operations as well as the signaling of exceptions, the requirement of exact rounding is somewhat loosened when dealing with decimal-binary and binary-decimal conversions.
- Finally, whereas we have been able to develop a set of test vectors that is completely precision independent when it comes to testing the operations (a), the testing of conversions is partly based upon different test vectors for different precision ranges. Full details can be found in [Verdonk et al. 2000].

The rest of the paper is structured as follows. Section 2 summarizes existing tools for testing implementations of IEEE floating-point arithmetic and gives an overview of the main features of our testing tool. In section 3 the concept of test vector and the precision independent syntax to encode floating-point operands, is introduced. This syntax is a streamlined and extended version of the syntax developed by [Coonen 1984]. In section 4 through section 8 we discuss, for each

of the six basic operations, which aspects of the operation are tested by the new test vectors. Section 9 describes the functionality of the driver program. In section 10, we discuss the results of applying our test tool to several implementations of standard and high precision floating-point arithmetic and indicate some directions for future work.

2. TOOLS FOR TESTING FLOATING-POINT HARDWARE

2.1 Existing tools

After publication of the IEEE standards for floating-point arithmetic [IEEE 1985] and [IEEE 1987], a number of tools were developed to test the correctness of a floating-point environment.

One of the first programs to test the quality of a floating-point implementation, is the so-called floating-point benchmark Paranoia [Karpinski 1985], originally written by W. Kahan. Paranoia determines which general properties are satisfied by a particular floating-point implementation e.g. what are the precision and exponent range, is the arithmetic rounded or chopped, is underflow gradual, is the implementation of square root monotone, is multiplication commutative, etc. Paranoia is now included in UCBTEST [Hough et al. 1988], which we shall discuss next.

UCBTEST is a whole set of programs for "testing certain difficult cases of IEEE floating-point arithmetic" [Hough et al. 1988]. As already mentioned, Paranoia is one of these programs. Three other programs, UCBmultest, UCBdivtest and UCBsqrttest, generate difficult test cases for multiplication, division and square root, respectively. These cases are obtained from number-theoretic algorithms developed by W. Kahan, as part of ongoing research into test methods for computer arithmetic. These cases are difficult in the sense that they yield results halfway or nearly halfway between representable numbers. Yet other programs in UCBTEST are designed to test the elementary functions, and we shall not discuss these here. The part of UCBTEST that is most relevant for our work is a battery of hexadecimal test vectors to test the basic operations $+, -, \times, /$ and $\sqrt{\ }$. For each of single, double and quadruple (16-byte format) precision, a separate, but analogous battery of test vectors is included in UCBTEST. Similarly, UCBTEST includes batteries of test vectors for several elementary functions in single, double and quadruple precision, but no vectors are provided for testing conversions nor for testing the IEEE remainder function. For a complete description of UCBTEST the reader is referred to [Hough et al. 1988].

Another commercial package, the NAG Floating-Point Validation package (FPV) [Du Croz et al. 1989], was developed for testing floating-point implementations, including the basic operations $+$, $-$, $\times$, $/$ and comparisons and, optionally, square root. It cannot test operations on denormalized numbers, nor the full range of exception handling facilities. FPV creates an extensive number of test operands by varying a limited number of floating-point patterns introduced by [Schryer 1981]. These patterns are the same for all operations. The rationale behind the patterns introduced by Schryer is that, [Du Croz et al. 1989]: " (...), especially in an implementation which is almost correct, errors are more likely to occur as edge effects, at or near some discontinuity or boundary in the values of the operands or some part of them.".

Last but certainly not least, a test suite which has been used by major manufacturers of IEEE hardware was developed by J.T. Coonen [Coonen 1984]. This test tool consists of a large database of test vectors together with a driver program. An essential feature of this tool is that the vectors are designed to be as format independent as possible. To run the tests, the driver program of [Coonen 1984] decodes the test vectors, given in a purposely designed syntax, into single, double and extended precision formats. Completely format-independent test vectors are included in this tool for the basic operations (a), while for most conversions (b) through (d), different conversion vectors are included for single, double and extended precision. For the conversions (e) no test vectors are included. When decoding the format independent set of test vectors from [Coonen 1984] into double precision representation, the intersection with the battery of hexadecimal double precision vectors from UCBTEST is rather large.

While formal verification methods have also been applied to floating-point systems, we shall not discuss such methods here but refer, among others, to [Russinoff 1998; Harrison 2000; German 1999; Cornea-Hasegan 1998].

2.2 Main features of the new precision and range independent tool

As one can see from the above overview, the available test tools are rather complementary in nature and each implement a different approach to testing: explicit testing of specific floating-point properties (Paranoia), explicit generation of difficult cases (UCBTEST), guided generation of bit patterns (FPV) and large databases of vectors to test several essential aspects of a floating-point implementation (Coonen in a precision independent format, and UCBTEST).

When constructing our precision independent tool, our goal was to combine as many features as possible of each of these approaches. We decided to opt for a driver program together with a very large set of precision independent test vectors.

- The first step in the development, was to classify and integrate the sets of test vectors from both [Coonen 1984] and [Hough et al. 1988] in a single suite and encode each test vector in a uniform way, where the precision and exponent range become parameters of the driver program.
- To this integrated set of vectors we added approximately 5500 new test vectors, more than doubling the set in size. Several of these new test vectors are precision independent encodings of hard cases, sometimes inspired by the programs in UCBTEST, and are described in the next sections.
- Following [Coonen 1984], we have fully documented the complete set of test vectors.
- For a specific exponent range $[L, U]$ and precision t, the driver program can convert the encoded, precision independent test vectors to an extended UCBTEST syntax. Hence, when the exponent range and precision correspond to one of the three floating-point formats supported by UCBTEST, compatibility is guaranteed with the UCBTEST suite of programs, if a minor update in the notation for the underflow exception is taken into account. A complete description of our extended UCBTEST format is given in Appendix B.
- Following the Paranoia philosophy, the test tool also reports which of the three IEEE-compliant underflow mechanisms is used by a floating-point implementa-

tion (see section 5.1) and checks whether that underflow mechanism is used consistently. Implicitly, also specific properties such as commutativity are checked.

- As for the guided generation of test operands according to certain patterns, we have developed, for our own purpose, a documentation syntax which allows to describe ranges (in significand and exponent field) of floating-point operands and corresponding results. Hence, by extending the driver program in the future, from a single test line in documentation syntax any combination of operands within a specific range (both for exponent and significand) can be generated. We have used this idea, which will be further discussed in section 10, to characterize the new test vectors in sections 4 through 8.
- Finally, as is detailed in section 9, the driver program has many options to test specific aspects of IEEE compliance, for example ignoring exception handling or only testing one of the four rounding modes required by the IEEE standard. This is especially relevant for multiprecision software implementations, most of which only support round to zero.

3. PRECISION AND RANGE INDEPENDENT TEST VECTORS

From section 2 it is clear that the main existing test sets for verifying implementations of IEEE floating-point arithmetic are [Coonen 1984] and [Hough et al. 1988]. Our first step in developing a precision independent tool for floating-point arithmetic was to classify and integrate the sets of test vectors from both these tools in a single suite and encode each test vector in a uniform, precision and range independent syntax. The syntax used by [Coonen 1984] seems to be a straightforward choice for this encoding. However, as pointed out in [Coonen 1984], this syntax is limited in that, roughly speaking, only 'simple' numbers modified in their low-order bits and possibly scaled up or down, can be represented. In order to enhance the test set, we needed to be able to encode much more general floating-point operands. Before describing the new, extended syntax, we briefly introduce some notations while reviewing the basic facts of IEEE floating-point arithmetic.

3.1 Notations and basic principles of IEEE floating-point arithmetic

Let $I\!F(\beta, t, L, U)$ be the set of floating-point numbers in base β, precision t and exponent range $[L, U]$. For every floating-point number $x \in I\!F(\beta, t, L, U)$, we shall denote its sign by $S_x \in \{+, -\}$, its significand by

$$s_x = \sum_{i=0}^{t-1} x_i \beta^{-i}$$

and its exponent by $e_x \in [L-1, U+1]$. In this paper we shall only consider the case $\beta = 2$. All floating-point numbers with exponent $e_x \in [L, U]$ are normalized ($x_0 = 1$) and their value is given by

$$S_x \times s_x \times \beta^{e_x} = S_x \sum_{i=0}^{t-1} x_i 2^{e_x - i}$$

The exponent values $e_x = L-1$ and $e_x = U+1$ are used to represent special values. If $e_x = L-1$, then the floating-point number is either ± 0 or the denormalized

number $\pm 0.x_1 \ldots x_{t-1} \times 2^L$, when at least one $x_i \neq 0$, $0 < i < t$. The special values $\pm\infty$ and NaN are encoded with exponent $e_x = U + 1$. For $\pm\infty$, we have $x_i = 0$ for $0 < i < t$, while for NaN, the fractional part $x_1 \ldots x_{t-1} \neq 0$ and is often used to store diagnostic information.

IEEE-754 requires that the exponent e_x of a floating-point number be stored in biased form. Hence, rather than storing the value of e_x, the biased exponent $e_x + B$ is stored, where the bias B is given by

$$B = U = -L + 1 \tag{1}$$

Since e_x varies between $L - 1$ and $U + 1$, the biased exponent varies between 0 and $2U + 1$.

In some instances, when x is a denormalized number, we shall need to refer not to the floating-point exponent $e_x = L - 1$ of x, but to the exponent E_x of the normalized representation of x. In other words, for any $x \in I\!F(2, t, L, U)$ we have

$$\begin{aligned} &\text{if} \quad e_x \in [L, U] \;:\; E_x = e_x \\ &\text{if} \quad e_x = L - 1\,,\; x_0 = \ldots = x_{j-1} = 0\,,\; x_j = 1 \;:\; E_x = L - j \quad 1 \leq j \leq t - 1 \end{aligned}$$

In general, any $x \in I\!R \setminus \{0\}$ can be written down as

$$x = S_x \sum_{i=0}^{+\infty} x_i 2^{E_x - i} \qquad \text{with } x_0 \neq 0 \tag{2}$$

Note that two such equivalent representations exist, one with only a finite number of nonzero x_i, the other with all but finitely many $x_i = 1$. Remember that for instance $\sum_{i=1}^{+\infty} 2^{-i}$ and 1.0 represent the same value.

To distinguish between the exact and the floating-point result of an operation, we shall use the notation $*$, respectively $\circledast$, where $* \in \{+, -, \times, /, \text{rem}\,, \sqrt{}\}$. To guarantee maximal accuracy, IEEE-754 requires that the operation $\circledast$ be implemented such that

$$x \circledast y = \bigcirc(x * y) \qquad \forall x, y \in I\!F(2, t, L, U) \tag{3}$$

where the rounding $\bigcirc : I\!R \to I\!F(2, t, L, U)$ satisfies the following properties:

$$\begin{aligned} \bigcirc(x) &= x && \forall x \in I\!F(2, t, L, U) \\ x \leq y \Rightarrow \bigcirc(x) &\leq \bigcirc(y) && \forall x, y \in I\!R \end{aligned} \tag{4}$$

Several roundings satisfy the properties (4). The IEEE standard requires that round to nearest $\bigcirc$, round to zero $\bowtie$, round up $\triangle$ and round down $\triangledown$ be supported. The conditions (3–4) guarantee that the relative error in the computed result is at most 1/2 ulp (Unit in the Last Place) of the computed result in round to nearest, and at most 1 ulp of the computed result in the other rounding modes, in the absence of overflow and underflow.

When rounding the exact result of an operation, the round and sticky bit play an important role and are denoted by ρ and σ respectively. Let the normalized exact result $z \in I\!R$ of an operation be of the form (2), then the round and sticky bit of z

are, with respect to the floating-point set $I\!F(2,t,L,U)$, given by:

$$\begin{array}{ll} \text{if } L \le E_z \le U: & \rho_z = z_t \qquad\qquad \sigma_z = (\sum_{i=t+1}^{+\infty} z_i \neq 0) \\ \text{if } E_z < L: & \rho_z = z_{t+E_z-L} \quad \sigma_z = (\sum_{i=t+1}^{+\infty} z_{i+E_z-L} \neq 0) \end{array} \tag{5}$$

In case $E_z > U$, the round and sticky bit are not relevant for rounding and hence left undefined. Here and in the sequel of the paper, we use the convention that $z_i = 0$ if $i < 0$ and also that $z_i = 0$ if $i \ge t$ and $z \in I\!F(2,t,L,U)$.

Apart from requiring exact rounding of the operations, IEEE–754 also requires that five types of exceptions be signaled when detected. According to the standard, the signal entails setting a status flag, taking a trap, or possibly doing both, the default being to respond without a trap. Hence, our test tool is designed to test exception handling in non–trapping mode. The five exceptions that can occur are invalid operation, division by zero, overflow, underflow and inexact. We refer the reader to [IEEE 1985] for a full description of the conditions under which each of these exceptions must be signaled. As the proper signaling of exceptions is also tested by our tool, we shall discuss in more detail each of the exceptions in sections 4 through 8. It should be mentioned that the only exceptions which can coincide are inexact with overflow and inexact with underflow.

3.2 The extended precision and range independent syntax for test vectors

Each test vector in our set essentially describes a floating-point operation: the operator, the rounding mode, the floating-point operands, the correctly rounded floating-point result as well as the exceptions raised by that operation. Because each floating-point number in a test vector is encoded in a precision and range independent way, the test vector can be used to test the operation in any floating-point set $I\!F(2,t,L,U)$, that is for arbitrary but fixed precisions t and exponent ranges $[L,U]$. The only conditions we impose are that

$$\begin{array}{c} t \ge 24 \\ U = -L + 1 = 2^{k-1} - 1 \qquad k \ge 8 \end{array}$$

Our syntax for the test vectors and for the encoding of floating-point numbers is a streamlined and extended version of the syntax in [Coonen 1984]. We shall now highlight the headlines of the extension. A complete description is given in Appendix A in BNF form. Let us start with an example test vector:

```
A+ =    1pt    4i1    1i2pt          A first, simple example
```

The leading character (A in this case) is the version number. This particular test vector tests the addition (+) in round to nearest (=) mode of the operands 1pt and 4i1. These operands respectively represent the floating-point numbers

$$1 \times 2^t \qquad (1 + 2^{-t+1}) \times 2^2$$

where t is the precision of the floating-point format under consideration. The operation raises the inexact exception (x) and the floating-point result is, in round to nearest, equal to 1i2pt or $(1 + 2^{-t+2}) \times 2^t$. The rest of the test line is comment.

In general, each test vector consists of at most 9 fields: version number and operator, precision specification (if applicable), rounding mode, first operand, second operand, exceptions, result and comment. The version number can be '2' or '3', indicating that the test vector is taken from Version 2, respectively Version 3, of the [Coonen 1984] test suite, 'H' indicating that it is taken from [Hough et al. 1988] while 'A' indicates that the test vector is new and was added by the authors. The operator is one of $+, \times, /$, % for remainder and 'S' for square root. The rounding mode is either $<, >, =$ or 0 for round down, round up, round to nearest and round to zero, respectively. Any combination of these rounding modes as well as the keyword 'ALL' are also syntactically correct. In that case the test vector is valid in all the specified rounding modes. For the exceptions, the abbreviations 'x' for inexact, 'o' for overflow, 'u', 'v' or 'w' for underflow (see section 5.1), 'i' for invalid and 'z' for divide by zero are used. If the operation raises no exceptions, rather than leaving the exception field blank, the keyword 'OK' is used for backward compatibility with [Coonen 1984].

In fact, none of the 9 fields making up a test vector, except the precision specification and the comment, may be left blank. The precision specification field was introduced by the authors and is almost always blank, except to indicate that the test vector is only valid for even ('e') or odd ('o') precisions. In the case of unary operators like square root, the value '0' is used as placeholder for the second operand.

The most important aspect of the new syntax is the way in which floating-point numbers are encoded. It is also in this aspect that the major extensions to the Coonen syntax have been introduced. The encoding of a floating-point number should be scanned from left to right and consists of an optional sign, a mandatory root number and zero or more modifier suffixes.

Root numbers are of several types: `S` for signaling NaNs, `Q` for quiet NaNs, `H` for infinity (think of Huge), `T` for the smallest normalized number (think of Tiny) and, of course, (exactly representable) integers. Hence, for example, all positive, normalized floating-point numbers lie in the interval [`T`,`H`[.

As in the original Coonen syntax, there are five so-called modifiers which can be used to modify root numbers. They each have the form <`oper`>`K`, where <`oper`> is one of five operators: `p` (plus), `m` (minus), `i` (increment), `d` (decrement) or `u` (ulp). However, whereas in [Coonen 1984] `K` is a digit between 0 and 9, it is much more general in our syntax:

- One can choose <`oper`>`K` equal to `pK` or `mK` to scale the root value up (`p`) or down (`m`) by 2^K where `K` is now a <`literal`>, not just a digit between 1 and 9. This literal can, of course, be one of the digits between 1 and 9, but also one of `t`, `h`, `B` or `B`<`digit`>. Here `t` is the precision of the floating-point set under consideration, `h`= $\lfloor$(`t` − 1)/2$\rfloor$ and `B` is the exponent bias as defined in (1). Furthermore, `B`<`digit`>= $\lceil$`B`/$2^{\texttt{<digit>}}\rceil$.
- One can choose <`oper`>`K` equal to `i`(<`pos`>)<`digit`> or `d`(<`pos`>)<`digit`>. If `t` is the precision of the floating-point set, then, in the original Coonen syntax, the first parameter <`pos`> is always implicitly equal to `t-1`, the last bit position in the significand of the root number. The modifiers `i`<`digit`> and `d`<`digit`> apply the function `nextfloat`, respectively `prevfloat`, to the root

number <digit> times, where nextfloat (prevfloat) returns the next (previous) representable floating-point number in the floating-point set $I\!F(2,t,L,U)$. Similarly, i(<pos>)<digit> and d(<pos>)<digit> encode the application, <digit> times, of the function nextfloat, respectively prevfloat, not to the root number but to the root number truncated after bit position <pos>, where <pos> can vary between 0 to t-1. The outcome of this operation is a number in $I\!F(2,$<pos>$+1,L,U)$, to which the remaining t-<pos>-1 bits of the original root number are appended to obtain a new floating-point number in $I\!F(2,t,L,U)$.

- Finally, one can choose <oper>K equal to u<digit> to replace the root value by <digit> units in its last place.

Hence K is generalized in two ways: it can either be a literal rather than just a digit, or it can be a bit position followed by a digit. The fact that K can be a literal rather than just a digit allows to modify the exponent range much more than was the case in the original Coonen syntax. The fact that K can be a bit position followed by a digit implies that it is now possible to modify not just low-order bits but also high-order bits or bits halfway the significand (depending on <pos>).

3.3 The complete set of precision independent test vectors

To construct our precision independent set of test vectors, the first step in the process was to integrate and classify the existing test set for fixed hardware formats from [Hough et al. 1988] with the set of vectors from [Coonen 1984]. The double precision vectors from [Hough et al. 1988] which were not in the Coonen test set, were encoded using the precision independent syntax just described. The vectors from [Coonen 1984] needed no or only minor modifications in their encoding. To this integrated set of vectors we added approximately 5500 new test vectors, upon which we shall comment in sections 4 through 8, more than doubling the set in size. It was indeed observed that in the set resulting from the merge of [Hough et al. 1988] and [Coonen 1984], not all combinations of round and sticky bit were tested systematically for each operation. Using the extended syntax, we were able to encode more general floating-point numbers and generate vectors to methodically test exact rounding. Several of the new test vectors are inspired by the philosophy underlying the UCBTEST and Paranoia testing tools.

4. ADDITION AND SUBTRACTION

Addition and subtraction are essentially the same operation, except for the sign of the second operand. Hence, test vectors for both operations are identical. The commutativity of addition is implicitly tested by the driver program, which reverses each test vector $x + y$ with $x \neq y$ to test $y + x$.

4.1 Exception handling

Of the five exceptions listed by the IEEE standard, only overflow, inexact and invalid can occur for addition and subtraction. Indeed, the underflow exception cannot arise for these operations when the floating-point number set includes denormalized numbers, as required by the IEEE standards. The invalid exception should only be raised when one of the operands is a signaling NaN or in case of magnitude subtraction of infinities. Test vectors with all possible combinations of

the special representations, such as quiet and signaling NaNs, $\pm\infty$ and ± 0, were included in the original Coonen set.

The overflow exception should be signaled when the exponent of the intermediate result, obtained by rounding the exact normalized result to t bits with the exponent range unbounded, exceeds the upper bound U of the exponent range. In the original [Coonen 1984] and [Hough et al. 1988] test sets, several operands x, y are included with $E_{x+y} > U$. In the new vectors for overflow testing, the operands are chosen such that the exact result $x + y$ equals

$$x + y = \pm(\sum_{i=0}^{t-1} 2^{U-i} + \sum_{i=t+j}^{2t+j-1} y_{i-t-j} 2^{U-i}) \qquad j \geq 0 \text{ fixed} \tag{6}$$

Depending on the rounding mode and on the value of the y_i's, the floating-point result $x \oplus y$ is then either infinity or the largest finite number (with the sign of the exact result). Only in the first case should overflow be signaled.

Case (6) could easily be encoded thanks to the introduction, in the extended syntax, of the literal `t` representing the working precision. Using the modifier suffix `mt`, it becomes possible to incorporate test vectors where the exponent of one operand is at least t less than the exponent of the other operand. For example, to generate case (6) with round bit 0 and sticky bit 1, the first operand is chosen to equal the maximal representable floating-point number $x = 2^{U+1}(1 - 2^{-t})$ while the second operand equals $y = 2^{U-t-1}$. Upward rounding of the exact result $x + y = \sum_{i=0}^{t-1} 2^{U-i} + 2^{U-t-1}$ results in $x \oplus y = +\infty$, while at the same time both the inexact and overflow exceptions should be signaled. In all other rounding modes, we have that $x \oplus y = \sum_{i=0}^{t-1} 2^{U-i}$ and only the inexact exception needs to be signaled. Similar test vectors are included for all possible combinations of round and sticky bit in case (6).

4.2 Exact rounding and the inexact exception

Using the extended syntax, we were able to create test vectors for many different alignments of the floating-point operands and at the same time generate all round and sticky combinations in the exact result $x + y$. For all cases in tables 1 and 2, the inexact exception should be signaled whenever the round bit ρ_{x+y} or the sticky bit σ_{x+y}, defined by (5), is non-zero.

In all cases in Table 1, the addition involves a pre-arithmetic shift of one of the operands. The bits of the shifted operand then completely determine the value of the round and sticky bit, except when subsequent post-arithmetic normalization due to carry propagation further influences the round and sticky bit. In Case 1 and Case 4, the alignment of the operands is such that the round and sticky bit are determined by the leading bits of the smallest (in absolute value) of the two operands. In Cases 2a–2b and 5a–5b, the round and sticky bit are determined by the trailing bits of the smallest (in absolute value) of the two operands. In Case 3, the computation of the exact result $x + y$ also involves carry propagation and hence post-arithmetic normalization.

The original [Coonen 1984] test suite also contains patterns which systematically test each bit in the computed result when the exponents of the two operands differ less than t. Of these test vectors, several were valid only for single and double

Table 1. Addition: exact rounding

	Operand x	Operand y	ρ_{x+y}	σ_{x+y}
Case 1	$(S_x, s_x, e_x \leq U)$	$(S_x, s_y, e_x - t - j \geq L)$ $j \geq 0$	y_{-j}	$\sum_{i=1-j}^{t-1} y_i \neq 0$
Case 2a	$(S_x, s_x, e_x \geq L)$	$(S_x, s_y, e_x + j \leq U)$ $1 \leq j \leq t-1$ $s_y = \sum_{i=0}^{j-1} y_i 2^{-i}$	x_{t-j}	$\sum_{i=t-j+1}^{t-1} x_i \neq 0$
Case 2b	$(S_x, s_x, L-1)$ $x_0 = 0$	$(S_x, s_y, L + j)$ $1 \leq j \leq t-1$ $s_y = \sum_{i=0}^{j} y_i 2^{-i}$	x_{t-j}	$\sum_{i=t-j+1}^{t-1} x_i \neq 0$
Case 3	$(S_x, \sum_{i=0}^{t-1} 2^{-i}, e_x)$ $e_x \leq U-1$	$(S_x, s_y, e_x - (t-j) \geq L)$ $j = 1, 2$	$(y_{j-1} + 1) \bmod 2$	$\sum_{i=j}^{t-1} y_i \neq 0$
Case 4	(S_x, s_x, e_x) $s_x \neq 1$	$(-S_x, s_y, e_x - t - j \geq L)$ $j \geq 0$	$(y_{-j} + \sigma_{x+y}) \bmod 2$	$\sum_{i=1-j}^{t-1} y_i \neq 0$
Case 5a	(S_x, s_x, e_x)	$(-S_x, s_y, e_x + j \geq L)$ $1 \leq j \leq t-1$ $\sum_{i=1}^{j-1} y_i \neq 0$	$(x_{t-j} + \sigma_{x+y}) \bmod 2$	$\sum_{i=t-j+1}^{t-1} x_i \neq 0$
Case 5b	$(S_x, s_x, L-1)$ $x_0 = 0$	$(-S_x, s_y, L + j)$ $1 \leq j \leq t-1$ $\sum_{i=1}^{j} y_i \neq 0$	$(x_{t-j} + \sigma_{x+y}) \bmod 2$	$\sum_{i=t-j+1}^{t-1} x_i \neq 0$

formats. For most of these precision dependent vectors that test shifting in addition, a precision independent counterpart is characterized in Case 6 in Table 2.

Test vectors were also added to check carry propagation caused by directed roundings.

5. MULTIPLICATION

In analogy with addition, the commutativity of floating-point multiplication is implicitly checked by generating for each new test vector, another one with the order

Table 2. Addition: operand shifting and exact rounding

	Operand x	Operand y	$x+y$	ρ_{x+y}	σ_{x+y}
Case 6	$(S_x, \sum_{i=0}^{j-1} x_i 2^{-i}, e_x)$ $e_x \leq U$ $0 < j < t$	$(S_x, s_y, e_x - j \geq L)$	$\sum_{i=0}^{j-1} x_i 2^{e_x - i} + \sum_{i \geq j} y_{i-j} 2^{e_x - i}$	y_{t-j}	$\sum_{i=t-j+1}^{t-1} y_i \neq 0$

of the arguments reversed. Also, all possible sign combinations are tested.

5.1 Exception handling

As for addition, the original Coonen set tests all possible combinations of special representations, including the cases where the invalid exception should be signaled.

Our contribution to the testing of overflow for multiplication consists in creating vectors where the exact result $x \times y$ of two floating-point operands x and y is only just larger than the largest representable floating-point number, namely

$$x \times y = S_x S_y (\sum_{i=0}^{t-1} 2^{U-i} + \sum_{i=t}^{2t-1} m_i 2^{U-i}) \tag{7}$$

The bits m_i determine the round and sticky bit. In the case of (7), correct rounding determines whether the floating-point result is signed infinity or the largest floating-point number with the sign of the exact result. At the same time, the overflow exception should, respectively should not be raised. These cases can be found in Table 3 when instantiating e_x and e_y such that $e_x + e_y = U$.

For the combination $(\rho_{x \times y}, \sigma_{x \times y}) = (1, 0)$, i.e. exact halfway cases, we found no solution of (7) which holds for all precisions or for all even precisions. If we choose floating-point operands $x = (S_x, 1 + 2^{-h} + 2^{-t+2}, e_x)$ and $y = (S_y, 1 + 2^{-1} + 2^{-2}, e_y)$ then $x \times y$ is an exact halfway case satisfying (7) only in case the precision t satisfies $t = 3k - 1$ for some $k \geq 2$. Because the precision of the most common hardware formats ($t = 24, 53, 64, 113$) does not satisfy $t = 3k - 1$, we have chosen not to include these vectors in the test set with a new precision specification.

According to the IEEE standard, two correlated events contribute to underflow: tininess and loss of accuracy. One can distinguish between three different ways to detect underflow, all of which are compliant with the standard. These three cases were labeled `u`, `v` and `w` in [Coonen 1984]. In order to characterize `u`-, `v`- and `w`-underflow, we introduce $result\_tmp(x * y)$, which is the normalized value obtained when rounding $x * y$ to t bits but as if the exponent range were unbounded. Here $*$ is either $\times$ or $/$ and $x * y$ is assumed normalized. We then have

$$\begin{aligned}
\texttt{u} - \text{underflow if} \ : \quad & |result_tmp(x * y)| < 2^L \\
& x \circledast y \neq result_tmp(x * y) \\
\texttt{v} - \text{underflow if} \ : \quad & |result_tmp(x * y)| < 2^L \\
& x \circledast y \neq x * y \\
\texttt{w} - \text{underflow if} \ : \quad & |x * y| < 2^L
\end{aligned}$$

Table 3. Multiplication: overflow exception

	t	Operand x	Operand y	$\rho_{x\times y}$	$\sigma_{x\times y}$
Case 7a		$(S_x, \sum_{i=0}^{t-3} 2^{-i}, e_x)$	$(S_y, 1+2^{-t+2}, e_y)$	1	1
Case 7b		$(S_x, \sum_{i=0}^{t-2} 2^{-i}, e_x)$	$(S_y, 1+2^{-t+1}, e_y)$	1	1
Case 7c	even	$(S_x, 1+2^{-h-3}+2^{-h-5}, e_x)$	$(S_y, \sum_{i=0}^{h+1} 2^{-i}+2^{-h-3}+2^{-h-4}, e_y)$	1	1
Case 7d	odd	$(S_x, 1+\sum_{i=h+2}^{t-1} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{h} 2^{-i}+2^{-t+2}, e_y)$	1	1
Case 8a	even	$(S_x, 1+2^{-h}, e_x)$	$(S_y, \sum_{i=0}^{h-1} 2^{-i}+2^{-t+2}+2^{-t+1}, e_y)$	0	1
Case 8b	even	$(S_x, 1+\sum_{i=h+2}^{t-1} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{h} 2^{-i}+2^{-t+2}, e_y)$	0	1
Case 9a	odd	$(S_x, 1+2^{-h}, e_x)$	$(S_y, \sum_{i=0}^{h-1} 2^{-i}+2^{-t+1}, e_y)$	0	1
Case 9b	odd	$(S_x, 1+\sum_{i=h+1}^{t-1} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{h-1} 2^{-i}+2^{-t+2}+2^{-t+1}, e_y)$	0	1
Case 10	odd	$(S_x, \sum_{i=0}^{h} 2^{-i}, e_x)$	$(S_y, 1+2^{-h-1}, e_y)$	1	0

$$x \circledast y \neq x * y$$

The implementor may choose to detect underflow according to one of these three criteria, but shall detect these events in the same way for all operations. Note that u-underflow is the situation where, due to an extraordinary denormalization loss (tininess), an error occurs (loss of accuracy) which exceeds the expected rounding error of 1/2 ulp of the computed result for round to nearest and of 1 ulp of the computed result for directed roundings.

Besides including new test vectors for underflow, a main new feature of the driver program is that it checks a floating-point implementation for consistent use of the same underflow criterion throughout.

Additional test vectors for underflow have also been included where, depending on the rounding mode, different underflow criteria are satisfied. For example, choosing the floating-point operands $x = 2^{E_x} + 2^{E_x-1}$ and $y = 2^{E_y} + 2^{E_y-t+1}$ we have that

$$x \times y = 2^{E_x+E_y} + 2^{E_x+E_y-1} + 2^{E_x+E_y-t+1} + 2^{E_x+E_y-t}$$

If $E_x + E_y = L - 1$, then in round down and round to zero

$$\begin{aligned} result\_tmp(x \times y) &= (1 + 2^{-1} + 2^{-t+1}) \times 2^{L-1} \\ &\neq (2^{-1} + 2^{-2}) \times 2^{L} \\ &= x \otimes y \end{aligned}$$

while in round to nearest and round up $result\_tmp(x \times y) = x \otimes y$. The above implies that in round down and round to zero, the u-underflow criteria are satisfied, while in round up and round to nearest only the v-underflow criteria are satisfied. In the former case, every floating-point implementation should signal inexact and underflow. In the latter, signaling the underflow exception is optional.

Other new test vectors for underflow include cases where the exact result of the multiplication is just smaller than the smallest representable denormalized floating-point number, namely

$$x \times y = S_x S_y \sum_{i=0}^{2t-1} m_i 2^{L-t-i} \tag{8}$$

Depending on the value of the bits m_i and on the rounding mode, the floating-point result $x \otimes y$ should be equal to zero or to the smallest denormalized floating-point number. Since the conditions for u-underflow are fulfilled in (8), every floating-point implementation should signal both the underflow and the inexact exception. Case 11, which also involves carry propagation, is an example of (8).

Table 4. Multiplication: carry propagation and underflow exception

	Operand x	Operand y	$x \times y$	$\rho_{x\times y}$	$\sigma_{x\times y}$
Case 11	$(S_x, \sum_{i=0}^{t-1} 2^{-i}, e_x)$	$(S_y, 1+\sum_{i=1}^{t-1} y_i 2^{-i}, e_y)$ $\exists i : y_i \neq 0$ $e_x + e_y = L - t - 1$	$S_x S_y (1 + \sum_{i=1}^{2t-1} z_i 2^{-i}) \times 2^{L-t}$	1	1

5.2 Exact rounding and the inexact exception

For multiplication, each combination of round and sticky bit can be generated with and without propagation of a carry and with normal as well as denormalized operands and/or result. Some of these cases can be found in Table 5. We remark that the first two of them are difficult, in the sense that the result is nearly halfway between representable numbers (Case 12 with $j = 1$) or is very close to a representable number (Case 13). For the exact halfway cases in Table 5, we have explicitly listed bit $t - 1$ in the exact, normalized result $z = x \times y$, since this bit,

together with the round and sticky bit, must be taken into account when rounding to even. It should be clear that cases similar to those in Table 5 can be constructed for denormalized operands and/or results. We do not list them here, but have included them in the test set.

Table 5. Multiplication: exact rounding

	Operand x	Operand y	$(x \times y)_{t-1}$	$\rho_{x\times y}$	$\sigma_{x\times y}$
Case 12	$(S_x, \sum_{i=0}^{t-2} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{t-1} 2^{-i} - 2^{-j}, e_y)$ $j = 1, 2$ $e_x + e_y + 1 \geq L$		$j \bmod 2$	1
Case 13	$(S_x, \sum_{i=0}^{t-1} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{t-j} 2^{-i}, e_y)$ $j = 1, 2$ $e_x + e_y + 1 \geq L$		0	1
Case 14a	$(S_x, 1 + 2^{-1} + 2^{-2}, e_x)$	$(S_y, 1 + y_{t-4}2^{-t+4} + 2^{-t+3}, e_y)$ $e_x + e_y + 1 \geq L$	y_{t-4}	1	0
Case 14b	$(S_x, 1 + 2^{-1}, e_x)$	$(S_y, 1 + y_{t-2}2^{-t+2} + 2^{-t+1}, e_y)$ $e_x + e_y \geq L$	y_{t-2}	1	0

6. DIVISION

6.1 Exception handling

All possible combinations of special representations were included in the original sets of [Coonen 1984] and [Hough et al. 1988], some of which raise the invalid exception, others the 'division by zero' exception, yet others raise no exception at all.

As for addition and multiplication, we would like to characterize floating-point operands x and y which, when divided, generate an (inexact) result equal to the largest representable floating-point number followed by one or more nonzero bits, such that rounding determines whether overflow will occur or not. However, it follows from the following lemma, with $E_z = U$, that such floating-point operands do not exist.

LEMMA 1. *Given any two floating-point operands x and y in $\mathbb{F}(2, t, L, U)$ with*

$$z = x/y = S_x S_y \Big(\sum_{i=0}^{t-1} 2^{E_z - i} + \sum_{i=t}^{+\infty} z_i 2^{E_z - i}\Big) \tag{9}$$

Then $z_i = 0$ for all $i \geq t$. Here E_z is either $E_x - E_y$ or $E_x - E_y - 1$.

Proof: Assume x, y in $I\!F(2, t, L, U)$ are such that (9) holds. We can write

$$x = S_x \sum_{i=0}^{t-1} x_i 2^{-i} \times 2^{E_x} = S_x \times \tilde{x} \times 2^{E_x}$$
$$y = S_y \sum_{i=0}^{t-1} y_i 2^{-i} \times 2^{E_y} = S_y \times \tilde{y} \times 2^{E_y}$$

We first consider $\tilde{x} < \tilde{y}$, in which case $E_z = E_x - E_y - 1$. Then, from (9),

$$\sum_{i=1}^{t} 2^{-i} < \tilde{x}/\tilde{y} < 1 \tag{10}$$

If we let

$$c = \tilde{y} \sum_{i=1}^{t} 2^{-i} = \tilde{y}(1 - 2^{-t}) = \tilde{y} - \tilde{y}2^{-t}$$

then the inequalities (10) are satisfied if and only if

$$\tilde{x} \geq \triangle(c) = \triangle(\tilde{y} - \tilde{y}2^{-t}) = \tilde{y}$$

which contradicts the assumption that $\tilde{x} < \tilde{y}$. The case $\tilde{x} > \tilde{y}$ is completely similar. □

Underflow test cases where the exact result is either smaller or larger than the smallest denormalized number are included in the test set. Examples are given in Table 6. If either the round or the sticky bit are different from zero in Case 15, the inexact and the underflow exception must be signaled by all IEEE-compliant implementations. It is indeed clear that the inexactness in Case 15 is only caused by denormalization loss and not by the fact that the exact result is not representable in t bits precision. Case 16 is such that, depending on the rounding mode, either the conditions for `u`-underflow (in round up and round to nearest) or `v`-underflow (in round down and round to zero) are satisfied. Note that for division, any result suffering `w`-underflow also suffers `v`-underflow. Indeed, it follows from Lemma 1 that if the exact, normalized result x/y is tiny before rounding, then it is also tiny after rounding.

6.2 Exact rounding and the inexact exception

Typical for the division of two floating-point numbers x and y, is that exact halfway cases, which can be difficult to detect, cannot occur as long as $|x/y| \geq 2^L$. Indeed, multiplying a halfway case result expressed in $t + 1$ bits (where t is the precision) with its floating-point divisor in t bits, would require at least $(t + 1)$ bits for the original floating-point dividend, which is not possible. More generally, the result of a floating-point division, when inexact, cannot be represented by a finite sequence of bits. As a consequence, an exact result $z = x/y$ with $E_z \in [L, U]$ and with a nonzero round bit, always has a non-zero sticky bit. This information is helpful when implementing a division algorithm which is based on the bitwise computation of the result.

Table 6. Division: underflow exception

	Operand x	Operand y	$z = x/y$	$\rho_{x/y}$	$\sigma_{x/y}$
Case 15	$(S_x, \sum_{i=0}^{3} 2^{-i} + \sum_{i=t-2}^{t-1} 2^{-i}, e_x)$	$(S_y, \sum_{i=0}^{1} 2^{-i}, e_y)$ $e_y \geq L$ $e_x - e_y = L - j$, $j \geq 1$	$S_x S_y(2^0 + 2^{-2} + +2^{-t+2}) \times 2^{e_x - e_y}$	z_{t-j}	$(\sum_{i=t-j+1}^{t-1} z_i \neq 0)$
Case 16	$(S_x, \sum_{i=0}^{t-1} 2^{-i} - 2^{-1}, e_x)$ $e_x \geq L$	$(S_y, 1 + 2^{-1}, e_y)$ $e_y \geq L$ $e_x - e_y = L - 1$	$S_x S_y(\sum_{i=0}^{t-2} 2^{-i} + + \sum_{i=0}^{+\infty} 2^{-t-2i}) \times 2^{L-1}$	0	1

Several tricky divide cases are included in the original Coonen test set, based on the simple power series expansion $x/(1 + N \cdot 2^{-t+1}) = x \times (1 - N \cdot 2^{-t+1} + O(N^2 2^{-2t+2}))$ [Coonen 1984]. These cases, which can nicely be encoded in a precision independent way, are difficult in the sense that the result is very close to a representable number (e.g. when $x = 1$ and $N = 1$) or nearly halfway between representable numbers (e.g. when $x = 1$ and $N = -1/2$). Such vectors are especially relevant for testing iterative division algorithms. Based on the same power series expansion, we have added a small number of precision independent test vectors, mainly with denormalized operands.

7. SQUARE ROOT

7.1 Exception handling

Neither underflow nor overflow can occur when computing the square root, while the original Coonen test set already contains several vectors to test the correct detection of the invalid operation exception when the operand is negative. Also all special representations are included as operand.

7.2 Exact rounding and the inexact exception

Halfway cases cannot occur as result of a square root operation. The representation of the square of such a halfway result would indeed require at least $2t + 1$ bits and this can never equal the original t-bit floating-point operand. More generally, one can state the following, in analogy with division.

LEMMA 2. *Let $x \in I\!F(2, t, L, U)$. There does not exist an integer $M \geq t$ such that*

$$z = \sqrt{x} = (\sum_{k=0}^{t-1} z_k 2^{E_z - k} + \sum_{k=t}^{M} z_k 2^{E_z - k})$$

with $z_M = 1$.

As explained in [Kahan 1996] the tricky part of iterative square root algorithms is getting the last rounding error right by less than 1/2 ulp of $\sqrt{x}$. A slight fumble during the square root computation can result in an error which exceeds 1/2 ulp so rarely, that random testing has practically no chance of exposing the flaw. Therefore, we have included test vectors for square root which are precision independent encodings of hard cases generated by the program UCBsqrttest [Hough et al. 1988]. They can also be constructed by considering the power series expansion

$$\sqrt{(x + N \cdot 2^{-t+1+e}) \times 2^{2\ell}} = (\sqrt{x} + \frac{1}{2}x^{-1/2}N \cdot 2^{-t+1+e} - \frac{1}{8}x^{-3/2}O(N^2 2^{2(-t+1+e)})) \times 2^{\ell} \tag{11}$$

where $e = \lfloor \log_2 x \rfloor$. For example, for $x = 1$ and $N > 0$ and even, the result is a nearly representable number, while for $N > 0$ and odd, the result is nearly halfway between representable numbers. The original Coonen test set already includes a very small number of test vectors based on the series expansion (11) with $x = 1$ and $\ell = 0$. Using the modifiers `pB<digit>` and `mB<digit>` we were able to encode square root operands and results with a wide range of exponents ℓ. Also, new test vectors satisfying (11) have been included, where the root number x equals y^2 for integers $y > 1$ and N an integer multiple of y.

The set of hexadecimal test vectors for square root in UCBTEST also includes random patterns which were not retained because of their precision-dependent nature.

8. REMAINDER

As indicated in IEEE-754, the remainder operation is not affected by the current rounding mode, and always delivers an exact result. The invalid exception can only occur in operations on NaNs.

In the original [Coonen 1984] test set for remainder, vectors are given for single, respectively double precision. As observed by J. Coonen, these vectors are in fact valid for all even, respectively odd precisions. We have included them in the test set for remainder using the precision specifier 'e'/'o' of our extended syntax. No essentially new test vectors for remainder have been constructed.

9. THE TEST DRIVER PROGRAM

So far, our discussion has focused on the test vectors. We shall now describe the driver program of our test tool. The purpose of the driver is twofold:

- translation of the generic test operands and result into binary representation, according to the precision and exponent range specified by the user.
- execution of the operation in each test vector by the implementation to be tested, and comparison of the result and exceptions with the given test vector; if applicable, performance of a commutativity check.

After execution of both phases, the driver program generates a log-file with the outcome of the testing, listing any errors that have occurred. Several options are available in both phases. For example, the tester can specify that the driver program should translate the test vectors, given in (extended) Coonen format, into UCBTEST hexadecimal format (see Appendix B) and output this to file rather

than perform actual testing (-o outputfile). The driver program can also take test vectors in UCBTEST format as input (-u inputfile). The options to specify the precision and exponent range of the binary floating-point set are:

`-e E`	provide E bits to represent the exponent
`-t T`	provide T bits precision
`-h`	**h**ide leading bit
`-s`	**s**ingle precision and exponent range (same as -e 8 -t 24 -h)
`-d`	**d**ouble precision and exponent range (same as -e 11 -t 53 -h)
`-l`	**l**ong double or extended precision and exponent range (same as -e 15 -t 64) as e.g. on Intel platforms
`-q`	**q**uadruple precision and exponent range (same as -e 15 -t 113 -h) as e.g. on Sun Sparc platforms
`-m`	**m**ultiprecision format (same as -e 15 -t 240)

Some of the other options influence the actual testing phase. For example:

`-n {o u x z i}`	do **n**ot test the listed exceptions (**o**verflow, **u**nderflow, ine**x**act, **z**ero divide, **i**nvalid)
`-r {p m n z}`	test only the listed **r**ounding modes (to **p**lus ∞, to **m**inus ∞, to **n**earest, to **z**ero)

The complete list of options is documented in the file `readme.usage`, which can be downloaded from `http://win-www.uia.ac.be/u/cant/ieeecc754.html`. On that same Web page, the source code of the testing tool, documentation on its use and the files with test vectors are also available.

To test a particular floating-point implementation, say X, one needs to write conversion routines between the internal representation of the floating-point number in X and the binary representation of that floating-point number in the driver program. A framework for these conversion routines is provided with the test tool software and, as will be discussed in the next section, the driver program comes with several example conversions for existing hardware and multiprecision software floating-point implementations.

10. APPLICATION, CONCLUSIONS AND FUTURE WORK

To illustrate the applicability and usefulness of our tool, we have applied it on one hand to hardware floating-point implementations, ranging from the Intel Pentium processors to Sun Sparc stations, and on the other hand to the multiprecision software library FMLIB [Smith 1991] and our own fully IEEE compliant multiprecision floating-point implementation MpIeee. Log files of these different tests are available on line at `http://win-www.uia.ac.be/u/cant/ieeecc754.html`.

On the Intel Pentium processor, which is an extended-based architecture, we applied our test tool to the native extended floating-point format (`-e` 15 `-t` 64), as well as to the IEEE single and IEEE double formats, which are supported by setting the rounding precision appropriately. As could be expected, our test tool did not report any error in extended floating-point arithmetic. Furthermore, it diagnosed the implementation of the `v`-underflow strategy. However, in single and double precision a change in the underflow strategy, as well as double rounding in some underflow cases, were detected by our test tool. It should be observed that

the double rounding cases are not in conflict with IEEE-754. Indeed, IEEE-754 requires that rounding precision be supported to allow systems, whose destinations are always extended, to mimic the precisions of systems with single and double destinations, but only in the absence of underflow and overflow. A detailed discussion of how these erroneous underflow cases arise, and of underflow detection in general, can be found in [Cuyt et al. 2000]. Finally, it should be observed that when testing the square root operation in extended precision by a call to the C/C++ function `sqrtl` using the GNU compiler gcc v2.95.2, superfluous exceptions are raised. The errors occur in cases where only the invalid exception should be signaled, while sometimes also the inexact or the overflow exception or both are signaled. When replacing the call to the function `sqrtl` by the appropriate assembler instruction, these errors disappear. This emphasizes the point made in section 1, concerning the binding between operators at the language level and the underlying hardware implementation, and the influence of compilers on this binding.

On the Sun SuperSparc and UltraSparc, which are both single/double-based architectures, we applied our test tool to the hardware single and double formats and the quadruple precision floating-point arithmetic available in software [SUN Microsystems 1997]. For all these formats, no errors were reported. Furthermore, our test tool diagnosed that in single and double precision, the `v`-underflow strategy is implemented on the Sun SuperSparc, while the `w`-underflow strategy is implemented on the more recent Sun UltraSparc. In quadruple precision, our test tool diagnosed that the `w`-underflow criterion is applied on both the SuperSparc and the UltraSparc processors.

We also applied our test tool to the well-known multiprecision library FMLIB [Smith 1991], which is one of the few multiprecision packages which supports two rounding modes: round to zero and 'almost always' round to nearest. According to [Smith 1991], no more than one in a thousand results will be incorrectly rounded to nearest for random operands. By setting the base β equal to 2, FMLIB was tested for several precisions and exponent ranges, in both round to zero and round to nearest. Since the library does not fully support special representations, underflow exception handling and denormalized numbers, we excluded all test vectors involving such aspects (by calling the driver program with the appropriate options). On the remaining test vectors, our test tool reported errors both in round to nearest and in round to zero mode for addition, multiplication, division and square root (no IEEE compliant remainder function is supported by FMLIB). For round to nearest, the number of errors reported was much larger than the 1‰ figure cited in [Smith 1991], which can easily be explained by the fact that our test vectors are not uniformly distributed but specifically include difficult to round cases. There was, however, no obvious explanation for the errors in round to zero or truncation mode. After contacting the author, it became clear that both rounding modes are handled in the same way: select the number of guard digits to use, compute the operation at that higher precision and then apply the rounding rule. In other words, round to zero is in fact also 'almost always' round to zero. In the new release of FMLIB, the number of guard digits is being increased to ensure exact rounding of the basic operations in both rounding modes.

In the framework of the Arithmos project [Cuyt et al. 2000], a high-performance class library has been developed implementing fully IEEE compliant multiprecision

floating-point arithmetic, with a user defined presicion t and base $2 \leq \beta \leq 2^{24}$. The development of this library was motivated by the general need for floating-point arithmetic with high precision, while, when building an interval arithmetic library on top of the floating-point implementation, the high precision allows one to push the outward interval rounding error further back. Although there exist several highly performant and powerful multiprecision packages, among which [Bailey 1995], FMLIB [Smith 1991] and MPFR [Zimmermann et al. 2000], most have not included full compliance with the principles of IEEE-754 among their design goals. This is confirmed by the FMLIB test results. With the development of MpIeee, it is shown that a multiprecision implementation of floating-point arithmetic, which is both highly performant and offers correctly rounded results and exception handling in line with IEEE-754, is possible. Our test tool was used for the thorough testing of this implementation.

Finally, it should be clear that the vectors in the test sets are only specific instantiations of the vectors we have characterized in Tables 1 through 6. By developing a more general driver program, which automatically instantiates test vectors from these characterizations, covering in a systematic way the ranges specified for exponent and significand, much more extensive testing can become possible. Such an approach would enhance our test tool even further, applying the idea of [Schryer 1981] not to a limited number of patterns which is the same for all operations as in [Du Croz et al. 1989], but rather to patterns specifically designed to test certain tricky aspects of a floating-point implementation.

ACKNOWLEDGMENTS

For the construction of the test sets we received valuable help from several people. We specifically thank Jerome Coonen for his help with different aspects of his original driver program, and David Hough for his feedback on the UCBTEST package. Finally, we are also indebted to Johan Bogo and Tim Gevers for their support on the development of the test driver.

The first and second author are respectively Research Director and Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (FWO). The third author is supported by a grant from the Flemish institute for the promotion of Scientific and Technological research in Industry (Belgium) (IWT).

APPENDIX

A. EXTENDED COONEN SYNTAX IN BACKUS-NAUR FORM (BNF)

```
<test vector>::= <version><operation> <prec> <rounding> <fp> <fp>
                 <exceptions> <fp>
<version>::= <digit> | H | A
<operation>::= + | - | * | / | % | S
<prec>::= {e | o}
<rounding>::= ALL | 0 | < | > | = | 0< | 0> | =< | => | =0> | =0<
<exceptions>::= OK | x | xo | xu | xv | xw | i | z
<fp>::=  {<sign>}<root>{<suffix>}*
<sign>::= + | -
<root>::=  Q | H | T | {<digit>}+
```

```
<suffix>::= {p<literal> | m<literal>} {i<spec> | d<spec>} {u<digit>}
<spec>::= <digit> | (<pos>) <digit>
<pos>::= <literal>{+<digit>} | <literal>{-<digit>}
<literal>:: = <digit> | t | h | B | B<digit>
<digit>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

B. EXTENDED UCBTEST SYNTAX IN BACKUS-NAUR FORM (BNF)

```
<test vector>  ::= <op><format> <rounding> <compare> <exceptions>
                   <fp> {<fp>} <fp>
<op>  ::= add | sub | mul | div | mod | sqrt
<format> ::= s | d | l | q | <exp> <hidden> <prec>
<exp> ::= <digit>+
<hidden> ::= 0 | 1
<prec> ::= <digit>+
<rounding>  ::= n | p | m | z
<compare> ::= eq | uo
<exceptions>  ::= - | x | xo | xu | xa | xb | v | d
<fp>  ::=  { <hex><hex><hex><hex><hex><hex><hex><hex> }+
<hex> ::= <digit> |  a | b | c | d | e | f
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The extended UCBTEST format is used by our driver program to translate, for a specific floating-point format, the precision independent vectors in our test set to actual bit representations. It is an extension of the existing UCBTEST format [Hough et al. 1988]. In this extended syntax, each test vector consists of the operation to be tested; directly followed by the floating-point format: `s` for IEEE single, `d` for IEEE double, `l` for Intel double extended (10-byte format) and `q` for SUN quadruple (16-byte format); the rounding mode; how the result should be checked: `eq`, or bit-by-bit equality for all results except NaNs and `uo` or unordered for NaNs; the exceptions; followed by the hexadecimal representation (in groups of 8 hexadecimal numbers separated by a space character) of the binary floating-point operands and of the result of the operation.

The main extension of the already existing UCBTEST format is that the floating-point format is no longer restricted to `s`, `d` or `q`, but that arbitrary floating-point formats can be represented. The format can be specified by `exp` for the bit size of the exponent, `hidden` to indicate whether or not the leading bit is hidden and `prec` for the bit size of the significand. Following the (extended) Coonen syntax, we have used three different flags to denote the three IEEE-compliant underflow mechanisms (see also section 5.1): `xu` for cases satisfying the `u`-underflow conditions, `xa` for cases satisfying the `v`- but not the `u`-underflow conditions (since the `v` flag is used in the original UCBTEST syntax to denote invalid) and `xb` for `w`-underflow cases. This is a minor update of the original UCBTEST syntax, where `xu` is used for both the `u`- and `v`-cases combined, while `x?u` is used for all cases of `w`-underflow.

REFERENCES

BAILEY, D. 1995. A FORTRAN 90-based multiprecision system. *ACM Trans. Math. Software 21*, 379–387.

Bailey, D., Simon, H., Barton, J., and Fouts, M. 1989. Floating-point arithmetic in future supercomputers. *Internat. J. Supercomputer Appl. 3*, 3, 86–90.

Batut, C., Bernardi, D., Cohen, H., and Olivier, M. *User's Guide to PARI-GP*. Laboratoire A2X, Université Bordeaux I, France. http://www.parigp-home.de/.

Brent, R. 1978. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software 4*, 1, 57–70.

Coonen, J. 1984. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*. Ph. D. thesis, University of California at Berkeley, USA.

Cornea-Hasegan, M. 1998. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal* Quarter 2.

Cuyt, A. et al. 2000. *The Arithmos project*. University of Antwerp (UIA). http://win-www.uia.ac.be/u/cant/arithmos.

Cuyt, A., Kuterna, P., Verdonk, B., and Verschaeren, D. 2000. Underflow revisited. Submitted for publication.

Du Croz, J., Erl, M., Gardner, P., Hodgson, G., and Pont, M. 1989. Validation of numerical computations in Ada. NAG Technical Report TR2/89, The Numerical Algorithms Group Ltd. (NAG).

German, S. 1999. Introductions to the special issue on verification of arithmetic hardware. *Formal Methods in System Design 14*, 1, 5–6.

Granlund, T. 2000. *The GNU Multiple Precision Arithmetic Library*. Swox AB, Stockholm. http://www.swox.com/gmp.

Haible, B. *CLN, a Class Library for Numbers*. ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/gnu/cln.tar.z.

Harrison, J. 2000. Floating-oint verification in HOL Light: the exponential function. *Formal Methods in System Design 16*, 3, 271–305.

Higham, J., Nicholas. 1996. *Accuracy and stability of numerical algorithms*. SIAM, Philadelphia.

Hough, D. et al. 1988. UCBTEST, a suite of programs for testing certain difficult cases of IEEE 754 floating-point arithmetic. Restricted public domain software from http://netlib.bell-labs.com/netlib/fp/index.html.

IEEE. 1985. *ANSI/IEEE Std 754-1985, Standard for Binary Floating-Point Arithmetic*. Reprinted in ACM SIGPLAN Notices 22(2):9-25, 1987.

IEEE. 1987. *ANSI/IEEE Std 854-1987, Standard for Radix-independent Floating-Point Arithmetic*.

Kahan, W. 1996. A test for correctly rounded SQRT. http://http.cs.berkeley.edu/~wkahan/SQRTest.ps.

Karpinski, R. 1985. Paranoia: A floating-point benchmark. *Byte*, 223–235.

Moshier, S. 1989. *Methods and Programs for Mathematical Functions*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, USA.

Russinoff, D. 1998. A mechanically checked proof of IEEE compliance of the AMD-K7 floating point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 148–200.

Schryer, N. 1981. A test of a computer's floating-point unit. Computer Science Technical Report 89, AT&T Bell Laboratories, USA.

Smith, D. 1991. Algorithm 693: a Fortran package for floating-point multiple-precision arithmetic. *ACM Transactions on Mathematical Software 17*, 2, 273–283.

SUN Microsystems. 1997. The UltraSparc processor. Technology White Paper, SUN Microsystems, inc., http://www.sun.com.

Tommila, M. 2000. *APFLOAT, A C++ High Performance Arbitrary Precision Arithmetic Package*. Helsinki University of Technology. http://www.jjj.de/mtommila/apfloat.

Verdonk, B., Cuyt, A., and Verschaeren, D. 2000. A precision independent tool for testing floating-point arithmetic II: conversions. Technical report, University of Antwerp, Dept. Math. and Comp. Science. Submitted for publication.

Zimmermann, P. et al. 2000. *MPFR: a library for multiprecision floating-point arithmetic with exact rounding*. http://www.loria.fr/projets/mpfr/.